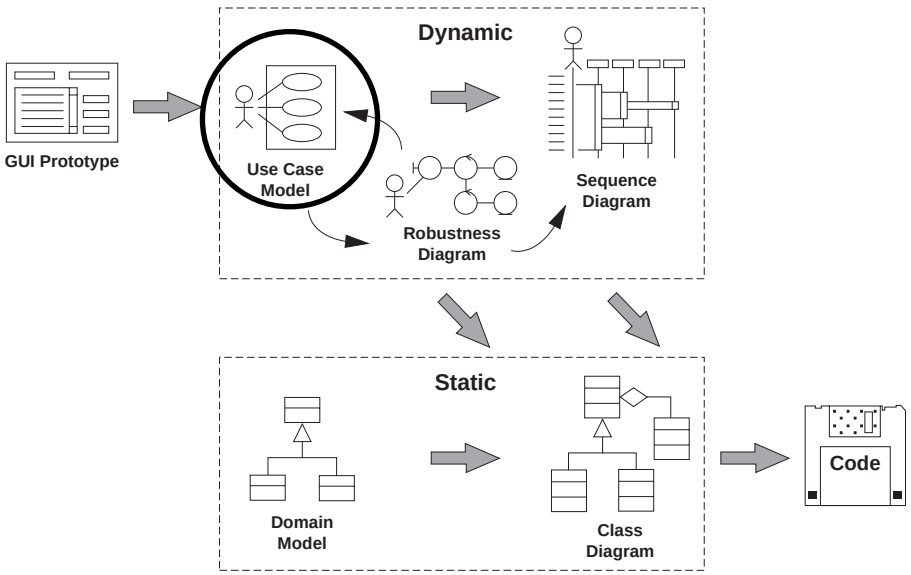
Chapter 3

Use Case Modeling

Within the Unified Object Modeling approach, one of the early steps involves building a **use case model**. The essence of this model is to capture user requirements of a new system, whether it's being developed from scratch or based on an existing system, by detailing all the scenarios that users will be performing.



RosenbergChapter3 Page 38 Friday, October 27, 2000 9:32 AM

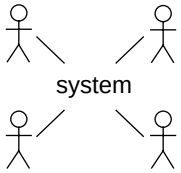


Figure 3-1 *Working Inward from User Requirements*

The use case model is at the conceptual center of the approach because it drives everything that follows, as you can see in the following list of the other key elements of the approach.

- The use case model is developed in cooperation with the domain model (Chapter 2).
- Robustness analysis (Chapter 4) involves identifying a first-cut set of collaborating objects that satisfy each use case.
- Interaction modeling (Chapter 5) is a refinement of the results of robustness analysis that lays out how messages flow between objects within use cases.
- Collaboration and state modeling (Chapter 6) involve exploring the dynamic behavior of objects within use cases.
- Requirements tracing (Chapter 7) involves connecting user requirements with use cases and classes.
- Use cases form the basis for user acceptance tests during the implementation phase (Chapter 8).

The *dynamic* model gets started with use case analysis. This effort involves working inward from the user requirements, as shown in Figure 3-1.

As you'll see from this point forward, use cases do more than get the dynamic model started—rather, they *drive* the dynamic model and, by extension, the entire development effort.

Use Cases, Actors, and Use Case Diagrams

A **use case** is a sequence of actions that an actor (usually a person, but perhaps an external entity, such as another system) performs within a system to achieve a particular goal.

RosenbergChapter3 Page 39 Friday, October 27, 2000 9:32 AM

A use case is most effectively stated from the perspective of the user as ***a present-tense verb phrase in active voice***. For instance, one use case within a hospital system might be called Admit Patient, while a portfolio system is likely to contain use cases named Do Trade Entry, Update Portfolio Information, and Generate Reports.

A complete and unambiguous use case describes one aspect of usage of the system *without presuming any specific design or implementation*. (However, it is a good idea to name those problem domain objects affected by the user's actions.) The result of use case modeling should be that *all* required system functionality is described in the use cases. If you don't adhere to this basic principle, you run the risk of having your bright engineers build a cool system that isn't what your customers want.

An **actor** represents a role a user can play with regard to a system or an entity, such as another system or a database, that will reside outside the system being modeled. The total set of actors within a use case model reflects everything that needs to exchange information with the system. Within a hospital system, actors include *Doctor*s and *Administrative Staff;* a portfolio system has actors called *Risk Manager* and *Trader.*

A user can serve as more than one type of actor. For instance, a *Nurse* might perform administrative duties. Similarly, more than one user can appear as a particular actor (for instance, multiple *Trading Assistants* interacting with a portfolio system).

We show use cases and actors on a **use case diagram**. Within a use case diagram, use cases appear as ovals, generally in the middle of the diagram; actors appear as stick figures to the left and right.

Figure 3-2 is an example of a use case diagram.

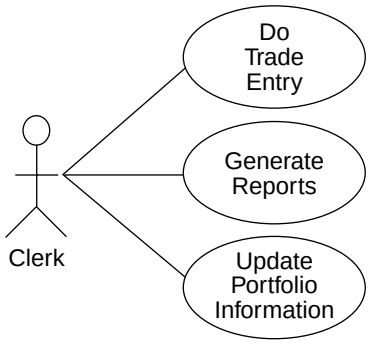


Figure 3-2 *Use Case Diagram*

RosenbergChapter3 Page 40 Friday, October 27, 2000 9:32 AM

Analysis-Level and Design-Level Use Cases

A key goal of use case driven object modeling involves identifying objects that can be reused throughout the system. Toward this end, the modeler can generate two types of use cases whose relationship parallels that of a class and an object belonging to that class.

An **analysis-level** (or **business process**) use case represents behavior that is common to a number of use cases. The term *analysis-level* has parallels with the term *abstract* in the context of C++, in that an analysis-level use case is never instantiated. In other words, it will not be used to directly drive an object model. (See Jacobson's *The Object Advantage: Business Process Reengineering with Object Technology* [Addison-Wesley, 1995] for more information.)

A use case that is instantiated is called a **design-level**, or **concrete**, use case. Design-level use cases sometimes use, or at least refer to, part or all of the description of the associated analysis-level use cases.

Take a typical word-processing program. It will have a menu bar option named Format. When the user selects that item, a window appears that offers a variety of choices that we can express in use case terms such as Format Font, Format Paragraph, and Adjust Tabs. Format represents the analysis-level use case; the three options are the design-level use cases. (You might also have a *package* of use cases, named Format, in this situation. I talk about use case packages later in the chapter.)

Writing Use Cases

Now that I've defined some basic terms and concepts, let's talk about how to write use cases.

We'll focus our efforts on design-level use cases, which some people call *scenarios*. You should be able to write a solid paragraph or two about a design-level use case. If you find yourself able to capture the essence of a proposed use case in, say, one sentence, it's likely you're breaking your system usage down too finely.

Just as the development process I describe in this book is an iterative and incremental process, the task of building use cases for your new

RosenbergChapter3 Page 41 Friday, October 27, 2000 9:32 AM

system is based on identifying as many as you can up front, and then establishing a continuous loop of writing and refining the text that describes them. Along the way, you will discover new use cases, and also factor out commonality in usage.

Working Inward from a GUI to Identify Use Cases

You should keep one overriding principle in mind at all times in your effort to identify use cases: ***They should have strong correlations with material in the user manual for the system***. It should be obvious what the connection is between each use case and a distinct section of your user guide. This reinforces the fundamental notion that you are conforming the design of your system to the viewpoints of the users. It also provides a convenient summary of what "use case driven" means: ***Write the user manual, then write the code.***

I encourage my clients to use rapid prototyping as frequently as possible. The idea is that developers and users sit down together and build something that will demonstrate "proof of concept."

I've found over the years that the best way to identify chunks of use cases in connection with a prototype is to write a rough user manual, as if the prototype were an actual fully working system. The manual certainly doesn't have to be polished; the idea is to describe the primary functions that users of the "system" perform, in language that's clear and unambiguous.

You will retrieve mostly basic courses of action from a prototype user guide, because by definition, "rapid" prototyping generally involves ignoring alternative courses (that is, exceptions). Be careful about this, though. After all, if all you had to build was a system that addressed basic courses, you'd probably finish ahead of schedule! (I'll reinforce this idea to a healthy extent later in the chapter.)

Taking this idea one step further, I've found that exploring the graphical user interface (GUI) design in parallel with the required system behavior is generally an excellent approach. This involves iterating, with the users, the presentation aspects of the system, and after achieving closure on a couple of screens, writing the associated use cases. This bouncing back and forth can be very effective in the right environment.

If prototyping isn't feasible or desirable in your situation, feel free to write science fiction. I mean that you should put *something* on

RosenbergChapter3 Page 42 Friday, October 27, 2000 9:32 AM

paper—screen mockups, pseudo-text, whatever—that will help you and your users understand each other and reach at least tentative agreement. It's not that important how close this something is to reality just yet.

If you're worried that prototyping might degenerate into extended GUI design, use line drawings like the one in Figure 3-3. This gives everyone a chance to focus on operational concepts, without getting into the hair-splitting that often occurs when people look at GUI designs. (The arrow to the right of the *Investment* field indicates the need for a drop-down menu.)

Here's how the associated use case might read:

> The trading clerk specifies the trade type. Then the clerk selects the investment involved in that trade from the list of available investments. The clerk also enters the ticket number that appears on the paper ticket for the order. Once all the necessary data is in place, the clerk presses a button to move to the next task.

Notice that the use case doesn't refer to the types of elements that appear in the mockup. The "radio buttons" could very well turn into a drop-down menu tomorrow. But because we've insulated the use case text from the details of the GUI, we've kept the spotlight on what the user needs to do with the system.

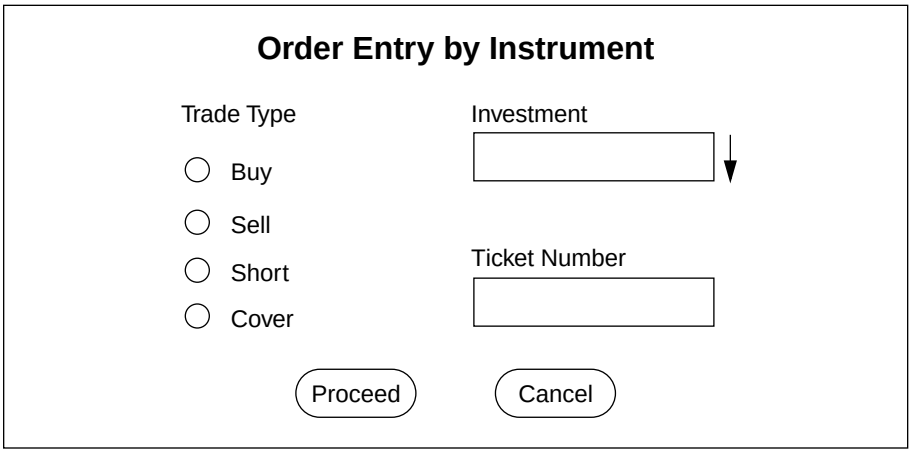


Figure 3-3 *Sample Screen Mockup*

RosenbergChapter3 Page 43 Friday, October 27, 2000 9:32 AM

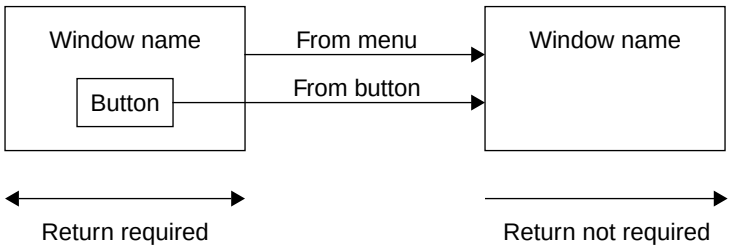


Figure 3-4 *Windows Navigation Diagram Elements*

Another useful approach to "GUI design without the GUI" comes from Meilir Page-Jones, author of the forthcoming book *Fundamentals of Object-Oriented Design in UML*. At the center of his approach is the **windows navigation diagram**, which he and his colleagues at Wayland Systems use extensively. The purpose of the diagram is to show how a user can move from one window to another along major, "application-meaningful" paths.

Figure 3-4 shows the basic elements of a windows navigation diagram.

Figure 3-5 shows an example of a windows navigation diagram. In this figure, the user arrives at a window on which she can modify a product price list. Selecting New from the File menu will take her to a

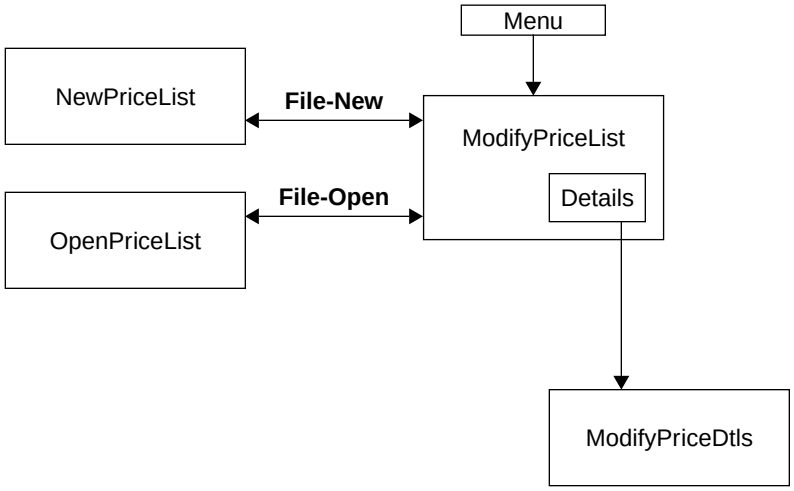


Figure 3-5 *Sample Windows Navigation Diagram*

RosenbergChapter3 Page 44 Friday, October 27, 2000 9:32 AM

window where she can start a new list; selecting Open from that menu will take her to a window where she can retrieve an existing list. Regardless of which path she takes, she must return to the main window, where she can press the Details button to bring up a "modify price details" window.

Whether you build a prototype, draw mockups, or use some other technique(s) to address the visual aspects of your system, it's very useful to link your screen designs to your use cases. You can do this manually or in the context of a visual modeling tool, such as Rational Rose (see Figure 3-6).

Think, again, in terms of a user manual. A screen shot appears, showing, for instance, your snazzy extra-wide scroll bars and unique button design. But the text doesn't say anything about that stuff—it just tells the reader what to do. For instance, here's what might appear in the user manual in connection with Figure 3-3:

1. Select the appropriate Trade Type.
2. Use the drop-down menu to choose the Investment involved in the trade you are entering.
3. Type the Ticket Number that appears on the paper ticket you are holding.
4. If you are satisfied with the entries you have made on this screen, click the Proceed button. Otherwise, click the Cancel button to return to the previous screen.

The coupling between the screen and the text is obvious, as is the similarity between this text and the use case text we've been presenting.

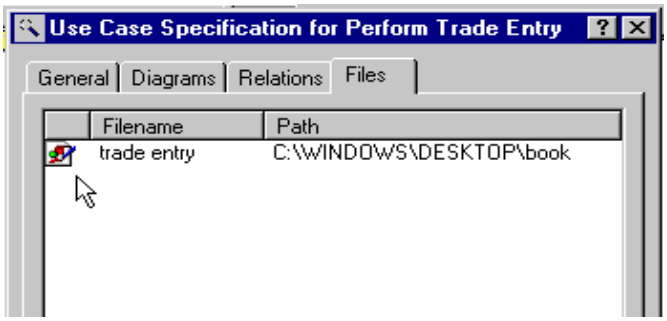


Figure 3-6 *Linking Files to Use Cases Within Rational Rose*

RosenbergChapter3 Page 45 Friday, October 27, 2000 9:32 AM

It's also true, though, that each piece can stand on its own. Just as a technical writer might choose to import that screen shot "by reference" instead of copying it directly into the manual, one might think of the use case model as incorporating the user interface model by reference.

I need to point out here that not only shouldn't your use case text include too many presentation details, but it should be relatively free of details about the fields on your screens, as well. ***Field names*** often match up directly with the ***names of attributes on your domain classes***, which I talk about in the next chapter. The narrative of a use case should be event-response oriented, as in, "The system does this when the user does that."

Mining Your Legacy User Manuals for Use Cases

The basic principles I described in this section don't apply only to newly conceived systems. ***If you're reengineering a legacy system, you can simply work from the user manual backward***. Look at the descriptions of the existing functionality. Then make changes based on how those functions will be performed in the new system. Instead of building a new manual from the ground up, you'll find yourself tearing the current manual down into its fundamental components, which will serve as the building blocks of your use case modeling.

For instance, suppose our example Portfolio Trading and Accounting system, which will have a GUI, was based on a system that had mainframe-style entry screens. Let's also suppose that the old trade entry screen for a bond trade involving a bond used function keys—F6 to go to a second entry screen that has other fields, for instance. In this case, the use case modeler would be able to replace the function key text with text that is less focused on the user interface and more focused on the desired system behavior.

Whether you use prototyping, screen mockups, or the results of mining legacy user manuals, it's important that you do a thorough job *before* you start writing use case text. If you don't, you could end up spending a *lot* of extra time trying to pin down what your users expect to be doing with the new system.

ALERT! *Don't* try *to write use cases until you know what the users will actually be doing.*

RosenbergChapter3 Page 46 Friday, October 27, 2000 9:32 AM

Refining Use Cases

We'll now start tracking the evolution of a use case for our new system, which we'll call Perform Trade Entry.

Here's a first cut:

> The Assistant Trader (AT) uses a trade entry window to enter the original data for a trade. The system validates the data before processing the trade. The AT has the option of making changes to trade data later.

This captures the basic points, but it's rather terse. When it comes to writing text for use cases, expansive is much preferable to terse. Let's see how we can improve and build upon this text.

Once you have some text in place for a use case, it's time to refine it by making sure the sentences are clear and discrete, the verbs are strong, and the actors and potential domain objects are easily identifiable. (Later in this chapter, I will talk about updating your static model with the results of your use case analysis. This is another recurring theme within the ICONIX approach: The static model isn't really static until the system works.) You are also likely to be in continuous learning mode as work progresses with your users, so you'll have a chance to add new knowledge to your text along the way.

Suppose we decided to combine the order entry task, which we discussed in "Working Inward from a GUI to Identify Use Cases" (see page 41), and the trade entry task we outlined within Perform Trade Entry. Here's how we might expand that use case's text:

> The Assistant Trader (AT) uses an order entry window to enter the data for an order. The system ensures that the trade number that the AT entered is unique. Then the system determines what type of investment is going to be traded and brings up the appropriate trade entry window in response. The AT then uses that window to enter the original data for the associated trade. When the AT finishes making entries, the AT indicates that the trade is ready for processing. The system validates certain entries (for example, it makes sure the settlement date does not precede the trade date) and then processes the trade appropriately.

RosenbergChapter3 Page 47 Friday, October 27, 2000 9:32 AM

As we walk through this use case, we can quickly classify the various parts of speech in modeling terms. There is a human actor named *Assistant Trader*. We can recognize objects named *Trade* and *Investment* that are already part of our domain model. And there will also be some kind of *Trade Entry Window*.

Chapter 4 describes the process by which you classify the objects you identify via use case analysis as boundary objects, entity objects, and control objects. Sometimes, it's actually easier to draw a robustness diagram before writing use case text. Regardless of which comes first, though, you need to keep in mind that ***you're not finished with a use case until the text and the robustness diagram match.***

Meanwhile, the latest version of Perform Trade Entry raises at least two questions.

1. What does "processing a trade" mean?
2. What happens if the AT does something wrong?

We'll deal with the first question when we come back to this use case in Chapter 5. The next section addresses the second question.

Basic and Alternate Courses of Action

A use case describes one or more *courses* through a user operation. The *basic* course must always be present; *alternate* courses are optional (but *not*, as we'll see, unimportant).

The **basic course of action** of a use case is the main start-to-finish path the user will follow, under normal circumstances (the "sunny day" scenario, in other words). For instance, within the Perform Trade Entry use case, all the text we have at this point is for the basic course, such as, "The system validates certain entries (for example, it makes sure the settlement date does not precede the trade date) and then processes the trade appropriately." There is an implicit assumption that the *Assistant Trader* has entered all the data for the given trade correctly—the date doesn't precede the current date, the price is expressed appropriately, and so forth.

RosenbergChapter3 Page 48 Friday, October 27, 2000 9:32 AM

An **alternate course of action** can represent an infrequently used path through the scenario, an exception, or an error condition. Here's an alternate course of action for Perform Trade Entry:

> If the validation fails, notify the user, highlight the erroneous values, and get new values from the user.

Basic courses of action are generally easier to identify and write text for. That doesn't mean, however, you should put off dealing with alternate courses until, say, detailed design. Far from it. In fact, all too often I've seen nasty errors of omission cause serious problems at that point in a project, when taking the time to write out alternate courses up front would have saved the team considerable grief.

It's been my experience that when important alternate courses of action are not uncovered until coding and debugging, ***the programmer responsible for writing or fixing the code tends to treat them in ways that are most convenient for him or her***. Needless to say, this isn't healthy for a project.

Although some authors encourage the use of voluminous use case templates, here's what I recommend to *every one* of my clients:

1. ***Create a use case template*** that has areas labeled Basic Course and Alternate Courses. Don't put anything else in there; it'll just distract you.
2. ***Ask "What happens?"*** This will get the basic course of action started.
3. ***Ask "And then what happens?"*** Keep asking that question until you have all the details of your basic course on paper.
4. ***Be relentless.*** "What *else* can happen? Are there *any* other things that can happen? Are you *sure*?" Keep asking *those* questions until you have a rich set of alternate courses written down. Trust me: Grief at this point is much easier to take than grief during, say, integration testing.

The goal is *not* to construct an elegant use case model; the goal is to account for *everything* the user might do.

You'll review this material before you finish use case modeling; you'll review it *again* during robustness analysis (discussed in the next chapter); and you'll review it *once more* during interaction modeling (see Chapter 5). This may seem excessive, but keep this in mind: The more well-defined the system behavior, the easier it's going to be to build the system.

RosenbergChapter3 Page 49 Friday, October 27, 2000 9:32 AM

Factoring Out Commonality in Usage

You can use several mechanisms to factor out common usage, such as error handling, from sets of use cases. This is usually a good thing to do, because breaking usage down to atomic levels will make your analysis effort easier, and save you lots of time when you're drawing sequence diagrams, too.

Constructs from the UML and OML

The UML offers three of these mechanisms.

1. **Generalization** works the same way with use cases as it does with regular classes: The child use case inherits the behavior and meaning of the parent use case, and so forth. The notation is the same, too. (See "Build Generalization Relationships" on page 21.)
2. The **includes** relationship (previously known as **uses**) involves one use case making full use of another use case. This goes to the issue of reuse, which is a prime objective of use case modeling, since it helps us factor out commonality. This construct is predefined as a **stereotype** that you can add to a use case diagram; you can also do what I do, and put "includes" as the label on the arrow.
3. The **extends** relationship allows the modeler to insert one use case into another use case, thus "extending" the second use case. The main idea behind this construct is that you can use it to show optional system behavior, or behavior that's executed only under certain conditions. This is also predefined as a stereotype.

Because the UML defines a use case as one form of a class, use case generalization is certainly a technically viable concept. However, I don't generalize use cases, because generalization is most useful for building elaborate structures dominated by abstract use cases.

These structures are great for organizing requirements that aren't linked directly to design elements, but we're talking about getting from use cases to code, so I won't talk anymore about use case generalization.

ALERT! ***Don't spend weeks building elaborate, elegant use case models that you can't design from.***

RosenbergChapter3 Page 50 Friday, October 27, 2000 9:32 AM

On the other hand, I *do* have something to say about *includes* and *extends*. Note that in the following discussion, I refer to the *includes* construct as *uses*, which is what *includes* was called in Jacobson's original book and in early versions of the UML.

In my years of teaching use case driven development, I have yet to come across a situation in which I have needed more than one mechanism for factoring out commonality. (I maintain this despite a seemingly endless debate about *uses* versus *extends* that is probably still going on, as you read this—albeit now in the form of *includes* versus *extends*—within the Object Technology User Group [OTUG]. See the appendix for more on this subject.)

I also think, however, that having two similar constructs is worse than having only one. It's just too easy to get confused—and bogged down—when you try to use both.

ALERT! ***Don't spin your wheels worrying about whether to use includes, extends, and/or uses.***

Even in the face of the terminology change within the UML, which doesn't address the fundamental problem, I find myself now leaning toward concepts from the Open Modeling Language (OML) called **invokes** and **precedes**. These would take the form of user-defined stereotypes on your use case diagrams.

The idea behind *invokes* is that one use case invokes another one in the same basic manner that a function invokes a peer function. You use *precedes* to indicate that one use case needs to precede another within a logical sequence.

The appendix offers samples of the long-running and often bitter debate between the advocates of *uses* and *extends* and the folks who are happy with *invokes* and *precedes*. Meanwhile, I use *invokes* and *precedes* in a couple of places later in this chapter.

One reason I like to see *precedes* on a use case diagram is that it helps keep the focus on what's inside the given use case. It's all too easy to get distracted by how you get there (that is, the **precondition**) and/or by what happens once you leave (the **postcondition**).

Once again, let me encourage you not to worry about adhering to the gospel of the UML when you do a project. Instead, you should use whatever will work well in a given situation.

RosenbergChapter3 Page 51 Friday, October 27, 2000 9:32 AM

Pick one of the mechanisms I've described here, or another one that you know about, and ***use it consistently***. Nor should you insist on using long and complex use case templates just because they appeared in a book or article.

ALERT! ***Don't waste time with long and involved use case templates.***

Consider the statements I just made as warnings about ways you might build up resistance to doing use cases, which offers an excuse to ditch modeling altogether—and we all know what happens then. (Hint: "The code's written, so I guess we're done!")

Back to Our Example

The current text for the basic course of action of our Perform Trade Entry use case, which appears at the bottom of page 46, is reasonable, but it's still pretty generic, and it also lends itself to factoring. Let's see how we can work with this text to create additional use cases, each of which is more specific—and thus more useful—for our example system. This is all part of our ongoing exploration of how the users will use the new system.

For starters, it turns out that buy trades and sell trades have certain features in common, as well as some different characteristics.

This leads us to do two things with Perform Trade Entry.

1. Create new use cases called Enter Buy Trade and Enter Sell Trade. These will capture the unique characteristics of the different types of trades.
2. Create a use case called Perform Order Entry that *precedes* either of the other new use cases. This will include the user behavior associated with the order entry task we originally decided to lump in with the trade entry task.

Figure 3-7 shows the results of performing these steps, in a use case diagram.

We can write the following text for the basic course of action of Perform Order Entry.

> The Assistant Trader (AT) uses an order entry window to enter the data for an order that involves a buy or sell trade. First, the AT specifies the trade type. Then the AT selects the

RosenbergChapter3 Page 52 Friday, October 27, 2000 9:32 AM

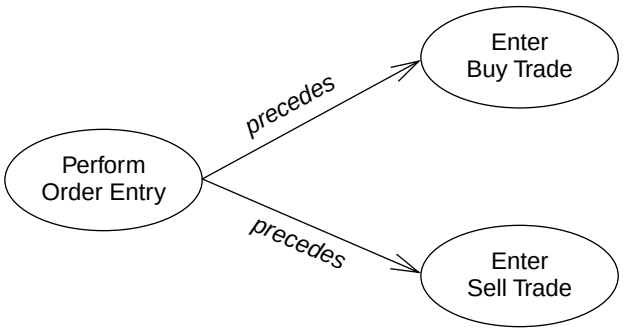


Figure 3-7 *Use Case Factoring*

investment involved in the order from a list of available investments. The AT also enters the ticket number that appears on the paper ticket for the order.

Once all the necessary data is in place, the AT presses a button to tell the system to process the order and bring up the appropriate type of trade entry window. The AT uses that window to enter the primary data for the trade connected with the new order.

The text for one alternate course of action is:

If the ticket number already exists, the system prompts the AT to enter a new ticket number.

Here is an attempt at another alternate course:

If the investment associated with an order is not present in the system, the AT brings up an investment entry window. The AT defines the appropriate information for that investment, and the system validates that information (for example, it ensures that the identifier appears in a known format). Once the system has accepted the investment definition, it returns the AT to the order entry window.

The problem with this text as an alternate course of action is that it introduces a new window that will likely have a number of different kinds of fields, in contrast with, say, an error dialog box. In situations like this, it's best to break out the text into another new use case. For our example, we'll call this new use case Define Investment.

RosenbergChapter3 Page 53 Friday, October 27, 2000 9:32 AM

We can use the preceding text as the basic course for Define Investment. For our alternate course, we can lift the text from our now-obsolete Perform Trade Entry use case:

> If the validation fails, notify the user, highlight the erroneous values, and get new values from the user.

With this new use case in place, the second alternate course for Perform Order Entry then becomes:

> If the investment associated with the order has not yet been defined, the system invokes the Define Investment use case.

In keeping with the idea of continuous improvement, the text for the basic course of action for Perform Order Entry enables us to compress the basic course of action for our new Enter Buy Trade use case. If we assume that the user wants to enter a trade that involves a purchase of bonds, the text might read:

> The Assistant Trader (AT) uses a Bond Trade Entry window to enter the primary values for the trade. The system validates both general trade values and bond-specific values (for instance, it makes sure the coupon rate is "reasonable") before processing the trade.

And we can reuse the alternate course for Define Investment as the alternative course for Enter Buy Trade:

> If the validation fails, notify the user, highlight the erroneous values, and get new values from the user.

At this point, we could define another new use case for error handling. Given the different sorts of business rules associated with *Trade*s and *Investments*, which we'll have to address somewhere down the line (during design), we won't go that route. Note, though, that it's always a good idea to keep your eye open for opportunities to simplify and clarify.

Before we proceed with the last two use cases that comprise our example system's use case model, notice that all the use case text in this section is in ***present tense and active voice***, and that the sentences are precise and clear.

This is the kind of text you would expect to see in a user manual. In fact, it's often a good idea to bring a technical writer into the process at

RosenbergChapter3 Page 54 Friday, October 27, 2000 9:32 AM

this stage for that very reason. A seasoned tech writer has a good feel for how to capture the essence of what a user wants to do. (Also, tech writers are not likely to be in a hurry to cut code.)

The text for the basic course of action for our new Enter Sell Trade use case, which is also *preceded* by Perform Order Entry, comprises the basic course text for Enter Buy Trade with an extra sentence in the middle to account for a key difference between buy and sell trades:

> The Assistant Trader (AT) uses a Bond Trade Entry window to enter the primary values for the trade. As part of this task, the system brings up a pairoff window, which the AT uses to select one or more buy trades to "pair off" with the new sell trade.
>
> The system validates both general trade values and bond-specific values (for instance, it makes sure that the coupon rate is "reasonable") before processing the trade.

I mentioned earlier that buys and sells have some common features. Therefore, we can use Enter Buy Trade's alternate course for Enter Sell Trade:

> If the validation fails, notify the user, highlight the erroneous values, and get new values from the user.

The last use case in our model has its roots in the following sentence from our original attempt at writing text for Perform Trade Entry: "The AT has the option of making changes to trade data later." (That one almost slipped through the cracks! ***Attention to detail is crucial!***)

This is the basic course of action for Perform Trade Adjustment:

> The Assistant Trader (AT) enters a trade number on a trade adjustment window. The system finds the data for the trade and displays it on the window. The AT can edit certain fields, such as the commission amount, but not others, such as the trade date and the quantity.
>
> After the AT has finished making entries, the AT submits the trade for processing. The system validates certain entries and then processes the trade adjustment appropriately.

The following are Perform Trade Adjustment's alternate courses:

> If the system cannot find the ticket number the AT entered, the system prompts the AT to enter a new ticket number.

RosenbergChapter3 Page 55 Friday, October 27, 2000 9:32 AM

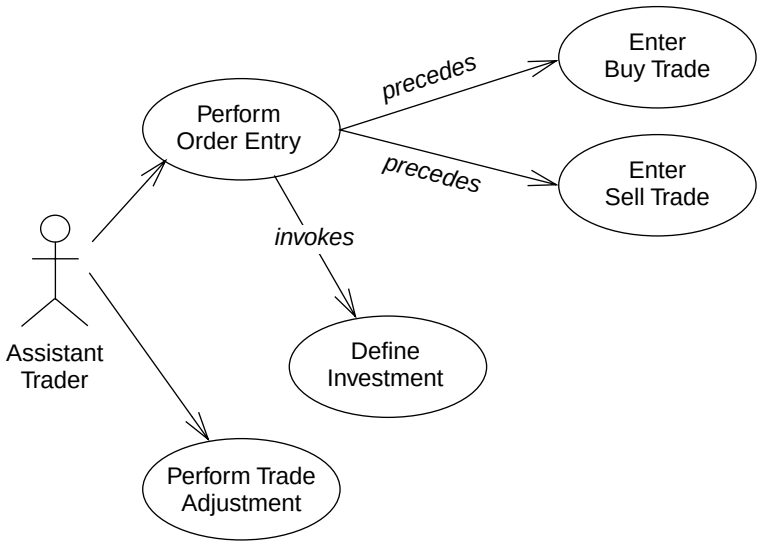


Figure 3-8 *Use Case Diagram for Example System*

> If the validation fails, notify the user, highlight the erroneous values, and get new values from the user.

Figure 3-8 shows the full use case diagram for our example system as it now stands.

Use Case Packages

In the UML, a **package** is a grouping of related elements, such as classes. I like to group use cases into packages, primarily because these packages form logical boundaries for dividing work among sub-teams. A good rule to follow is: ***Each package should correspond with a chapter, or at least a major section, in your user manual***.

Figure 3-9 is a **package diagram** that shows two packages for our example system: one that contains the use cases we've been looking at, and another that holds use cases that might appear in another part of the system. Note that you can put a class diagram in a package, too; a

RosenbergChapter3 Page 56 Friday, October 27, 2000 9:32 AM

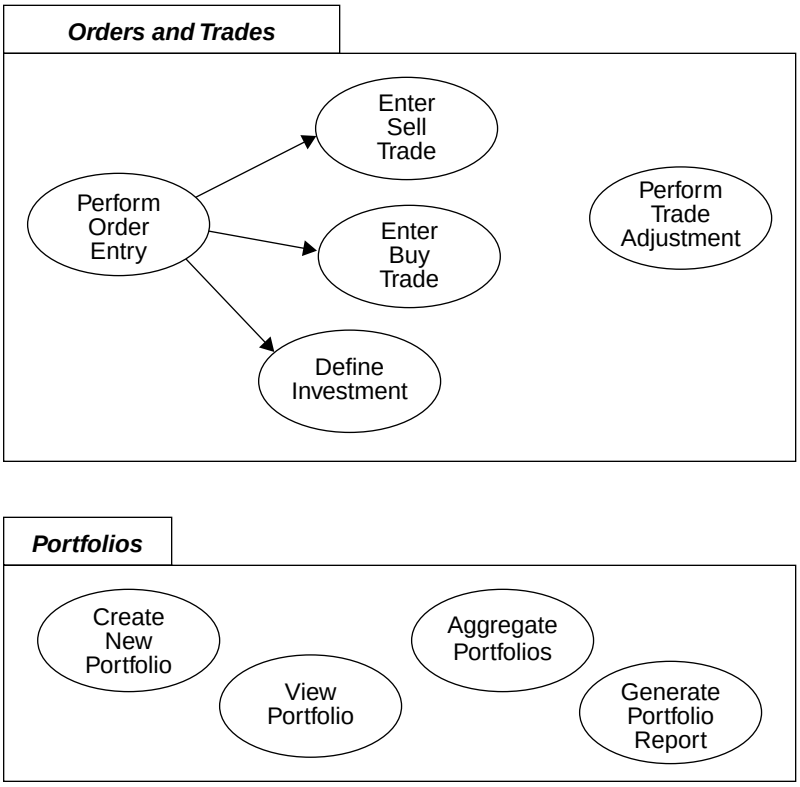


Figure 3-9 *Package Diagram for Example System*

package can contain any other element of the UML (including another package).

Use Cases and Requirements

The relationship between requirements and use cases is the subject of much discussion—well, argument—in the object-oriented community. My take on it can be summarized as follows.

RosenbergChapter3 Page 57 Friday, October 27, 2000 9:32 AM

- A use case describes a unit of behavior.
- A requirement describes a law that governs behavior.
- A use case can satisfy one or more functional requirements.
- A functional requirement may be satisfied by one or more use cases.

Note my use of the words *may* and *functional*. A system will have its share of functional requirements, but it will also have other types of requirements, such as those involving performance and maintainability, that won't map well to use cases. I explore different types of requirements and talk about the importance of traceability in connection with those requirements in Chapter 7.

Wrapping Up Use Case Modeling

You should feel comfortable proceeding to the next phases of the development process when you've achieved the following goals of use case modeling:

1. You've built use cases that together account for ***all*** of the desired functionality of the system.
2. You've produced clear and concise written descriptions of the basic course of action, along with appropriate alternate courses of action, for each use case.
3. You've factored out scenarios common to more than one use case, using the *precedes* and *invokes* constructs (or whichever constructs you're most comfortable using).

Figure 3-10 and Figure 3-11 show the tasks I discussed in this chapter.

I talk about requirements at some length in Chapter 7. As I indicate in Figure 3-10, you should perform at least a preliminary review of your requirements with your users before you proceed with robustness analysis, the subject of the next chapter. There's no sense heading into preliminary design unless you're still on track with your customers!

RosenbergChapter3 Page 58 Friday, October 27, 2000 9:32 AM

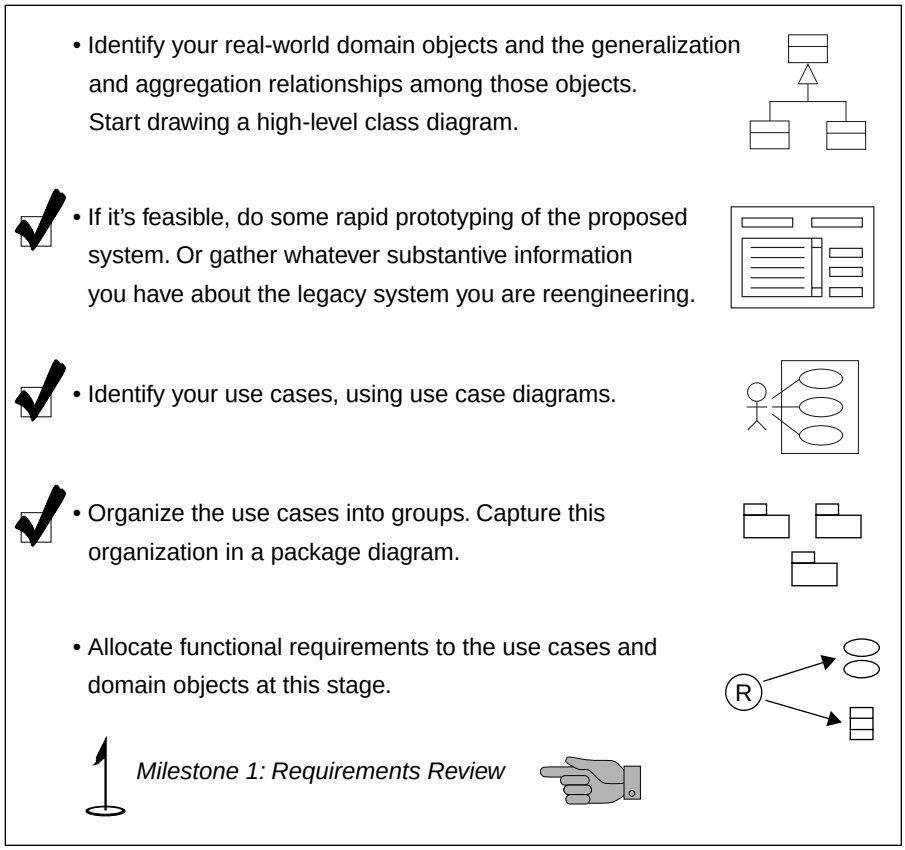
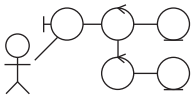

• Identify your real-world domain objects and the generalization and aggregation relationships among those objects. Start drawing a high-level class diagram.
• If it's feasible, do some rapid prototyping of the proposed system. Or gather whatever substantive information you have about the legacy system you are reengineering.
• Identify your use cases, using use case diagrams.
• Organize the use cases into groups. Capture this organization in a package diagram.
• Allocate functional requirements to the use cases and domain objects at this stage.
R
Milestone 1: Requirements Review

Figure 3-10 *Requirements Analysis Checkpoint 2*

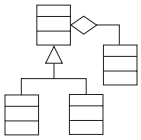
RosenbergChapter3 Page 59 Friday, October 27, 2000 9:32 AM

- Write descriptions of the use cases—basic courses of action that represent the "mainstream" and alternate courses for less-frequently traveled paths and error conditions.

- Perform robustness analysis. For each use case:
  - Identify a first cut of objects that accomplish the stated scenario. Use the UML Objectory stereotypes.
  - Update your domain-model class diagram with new objects and attributes as you discover them.



- Finish updating the class diagram so that it reflects the completion of the analysis phase of the project.



Milestone 2: Preliminary Design Review

Figure 3-11 *Analysis and Preliminary Design Checkpoint 1*

RosenbergChapter3 Page 60 Friday, October 27, 2000 9:32 AM

Top 10 Mistakes to Avoid When Writing Use Cases

10. *Write functional requirements instead of usage scenario text.*
9. *Describe attributes and methods rather than usage.*
8. *Write the use cases too tersely.*
7. *Divorce yourself completely from the user interface.*
6. *Avoid explicit names for your boundary objects.*
5. *Write using a perspective other than the user's, in passive voice.*
4. *Describe only user interactions; ignore system responses.*
3. *Omit text for alternative courses of action.*
2. *Focus on something other than what's "inside" a use case, such as how you get there (precondition) or what happens afterward (postcondition).*
1. *Spend a month deciding whether to use* includes *or* extends.